Observerables are not Fields

CS 5010 Program Design Paradigms
"Bootcamp"
Lesson 10.7
Key Points of this lesson

• Objects have some behaviors that are observable and some that are not observable.
• We can only test the observable behaviors of an object.
• We can create multiple classes that implement the same observable behavior in very different ways.
• An interface acts like an API; multiple implementations of an interface are like multiple implementations of an API.
Taking advantage of non-observables

• The set of observables in the problem set is purposely minimal, in order to give you the maximum freedom in implementing the objects.

• Let's see how you might take advantage of this.

• Here's a simple interface:
StupidRobot

;; A StupidRobot represents a robot moving along a one-dimensional line, ;; starting at position 0.

(define StupidRobot
  (interface ()

    ;; a new StupidRobot is required to start at position 0

    ;; -> StupidRobot
    ;; RETURNS: a Robot just like this one, except moved one ;; position to the right
    move-right

    ;; -> Integer
    ;; RETURNS: the current x-position of this robot
    get-pos
  ))

The only observable is the position of the robot.
Scenario and Observation

If we have a correct implementation of `StupidRobot<%>`, the following test should pass:

```scheme
(local
  ((define r0 ..a new StupidRobot<%>..)
   ;; move r0 right twice
   (define r1 (send (send r0 move-right) move-right)))
  ;; get-pos should then return 2
  (check-equal
   (send r1 get-pos)
   2))
```
The "obvious" implementation

(define Robot1%  
  (class* object% (StupidRobot<%>)  
  
    (init-field [x 0])  
    ;; interp: the position of the robot.  

    (super-new)  

    (define/public (move-right)  
      (new Robot1% [x (+ x 1)]))  

    (define/public (get-pos)  
      x)  

  ))

Here the observable is the value of a field.  
Of course, our choice of x for the field name is arbitrary.
You could name fields anything you want

(define Robot2%  
  (class* object% (StupidRobot<%>))

  (init-field [blerch 0])
  ;; interp: the position of the robot.

  (super-new)

  (define/public (move-right)  
    (new Robot1% [blerch (+ blerch 1)]))

  (define/public (get-pos)  
    blerch)
)

Of course, our choice of x for the field name was arbitrary. We could have named it anything we wanted, so long as we gave it a proper interpretation.
But we could have done it differently

(define Robot3%
  (class* object% (StupidRobot<%>)

    (init-field [y 0]) ;; interp: the negative of the position of the robot.

    (super-new)

    (define/public (move-right)
      (new Robot2% [y (- y 1)]))

    ;; RETURNS: the x-position of the robot
    (define/public (get-pos)
      (- y))

))

Here the observable is *not* the value of any field. The observation method translates the field value into the external value of the observable.
Or we could have done it very differently

\begin{Verbatim}
(define Robot4%
 (class* object% (StupidRobot<%>))

 (init-field [x empty])
 ;; Interp:
 ;; a list whose length is equal to the position of the robot

 (super-new)

 (define/public (move-right)
  (new Robot3% [x (cons 99 x)]))

 ;; RETURNS: the x-position of the robot
 (define/public (get-pos)
  (length x))
)
\end{Verbatim}

Puzzle: the other two implementations would work fine if we had a \texttt{move-left} method as well as \texttt{move-right}. How could you modify this implementation to handle \texttt{move-left}?
All three of these implementations have the SAME observable behavior

- no combination of scenarios and observations can tell them apart!
- If these are the only methods and observations we have on these objects, then we don't care which implementation we use— they will behave the same in any program.
- We could even write something like
Choose a random implementation

;; -> StupidRobot<%>
(define (new-robot)
  (local
    ((define i (random 3)))
    (cond
      [ (= i 0) (new Robot1%)]
      [ (= i 1) (new Robot2%)]
      [ (= i 2) (new Robot3%)]
      [ (= i 3) (new Robot4%)])

Returns a random number between 0 and 3
Signatures and Interfaces (again)

;; move-n : Robot<%> Nat -> Robot<%>
(define (move-n r n)
  (cond
   [(zero? n) r]
   [else (move-n
             (send r move-right)
             (- n 1))]))

This works with ANY class that implements Robot<%>.

Signatures should be in terms of interfaces, not classes.
This is important in practice

• We often deal with situations in which we have an interface that defines a set of operations and observations, and our program will work correctly no matter which implementation is underneath.

• Different implementations may have different performance characteristics, however.
Example: File System Interface

(define FileSystem<%>
  (interface ()
    open
close
read
write
; ... ))

• The interface to a file system is a list of procedures that you can use to create and manipulate files

• In other words, it's an API ("application program interface")
Example: File System Implementations

A file system, like NTFS, FAT, ext4, or the Google filesystem, is an implementation of the file system interface.

Different file system implementations will have different performance characteristics.
Example: File Systems

(define mydiskC
  (new NTFS%
      [ntfs-param1 'a-value1]
      [ntfs-param2 'another-value]))

(define my-network-fs
  (new GFS%
      [gfs-param1 'a-different-value]
      [gfs-param2 'yet-another-value]))

• On any particular computer system, we'll have some real file systems.
• Here my C disk is an NTFS file system, and I've also mounted a network file system that is a Google File System.
That's an analogy, folks...

• Most operating systems don't pretend a file system is an object.

• But they could, in principle.
  – Unix pretends that every device is a file (eg /dev/tty).
  – So one could imagine an operating system in which every resource is presented as an object.
Lesson Summary

• We've seen that
  – Objects have some behaviors that are observable and some that are not observable.
  – We can only test the observable behaviors of an object.
  – We can create multiple classes that implement the same observable behavior in very different ways.
  – An interface acts like an API; multiple implementations of an interface are like multiple implementations of an API.